

Exhibit N to the
Declaration of Imran A. Khaliq In Support
Of Visto's Opening Claim Construction
Brief Under P.R. 4-5(a)



US005600834A

United States Patent [19]

Howard

[11] Patent Number: 5,600,834
 [45] Date of Patent: Feb. 4, 1997

[54] **METHOD AND APPARATUS FOR RECONCILING DIFFERENT VERSIONS OF A FILE**

[75] Inventor: John H. Howard, Cambridge, Mass.

[73] Assignee: Mitsubishi Electric Information Technology Center America, Inc., Cambridge, Mass.

[21] Appl. No.: 417,446

[22] Filed: Apr. 5, 1995

Related U.S. Application Data

[63] Continuation of Ser. No. 61,674, May 14, 1993, abandoned.

[51] Int. Cl.⁶ G06F 15/00; G06F 7/00

[52] U.S. Cl. 395/617; 395/200.08; 395/200.19; 364/DIG. 1; 364/282.1; 364/281.5; 364/282.4

[58] Field of Search 395/600, 200.08, 395/200.19; 364/DIG. 1

[56] References Cited

U.S. PATENT DOCUMENTS

4,077,059	2/1978	Cordi et al.	364/200
4,408,273	10/1983	Plow	364/200
4,819,156	4/1989	Delorme et al.	364/200
4,823,310	4/1989	Grand	364/900
4,875,159	10/1989	Cary et al.	364/200
4,914,654	4/1990	Matsuda et al.	370/94.1
5,151,988	9/1992	Yamagishi	395/600
5,155,849	10/1992	Westfall et al.	395/600
5,212,788	5/1993	Lomet et al.	395/600

5,247,684	9/1993	Tavares et al.	395/700
5,261,094	11/1993	Everson et al.	395/600
5,263,155	11/1993	Wang	395/600
5,357,631	10/1994	Howell	395/600
5,388,255	2/1995	Pytlak et al.	395/600

Primary Examiner—Wayne Amsbury

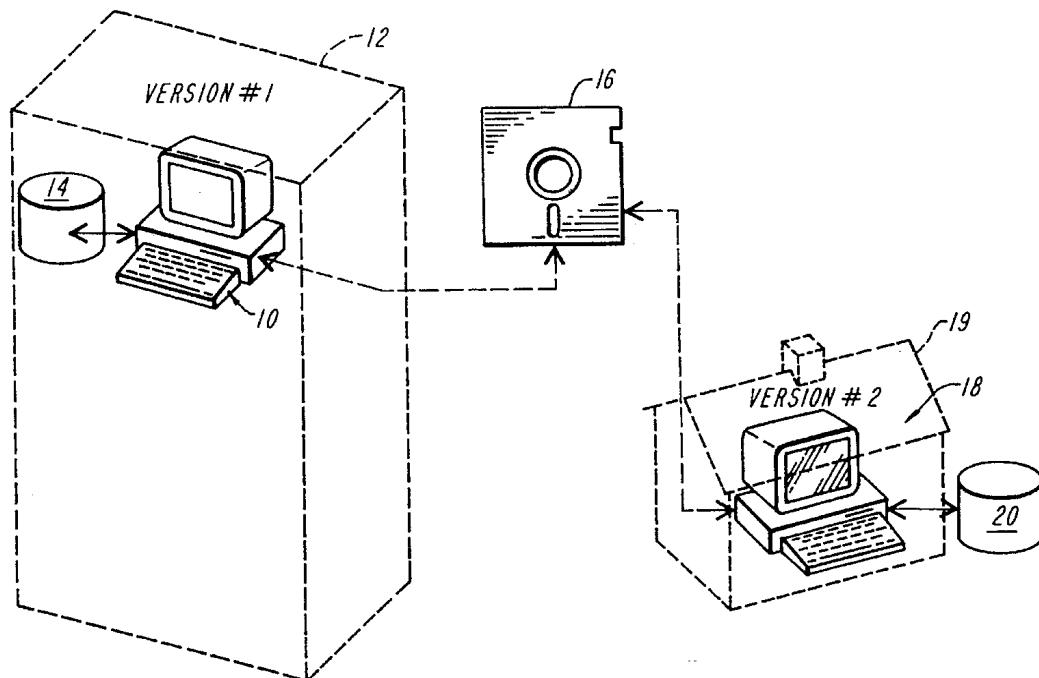
Assistant Examiner—Jean R. Homere

Attorney, Agent, or Firm—Robert K. Tendler, Esq.

[57] ABSTRACT

In a distributed file environment, a system for safely updating a file without risk of losing work performed at one site due to work performed on the file at another site uses a journal or log at each site which is updated after a file is modified. This log is compared with the logs from other sites before a file is used at any one site, so that new versions can be propagated automatically and safely to out-of-date sites, with the user immediately alerted if conflicting versions of the file exist at different sites. The reconciliation can be applied to collections of files, automatically updating only those files for which it is safe to and necessary do so. Since reconciliation occurs at times selected by the user, inconsistent or partially completed versions of files need not be propagated to other sites. Additionally, logs may be built incrementally by occasionally observing the state of the systems in terms of the files and their time stamps and creating additional log entries reflecting appearance, disappearance and changes of files. Furthermore, logs may be purged of obsolete entries by including additional log entries indicating the most recent time each site has participated in a reconciliation and deleting obsolete entries that all sites have seen.

4 Claims, 5 Drawing Sheets



U.S. Patent

Feb. 4, 1997

Sheet 1 of 5

5,600,834

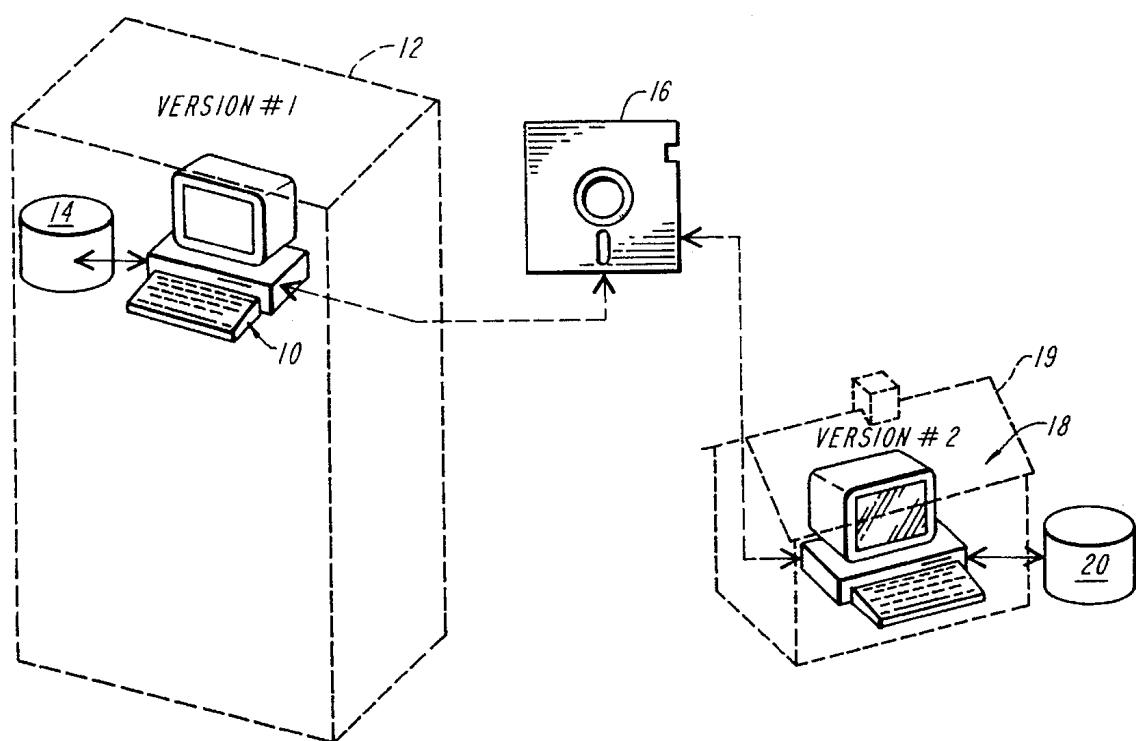


FIG. 1

U.S. Patent

Feb. 4, 1997

Sheet 2 of 5

5,600,834

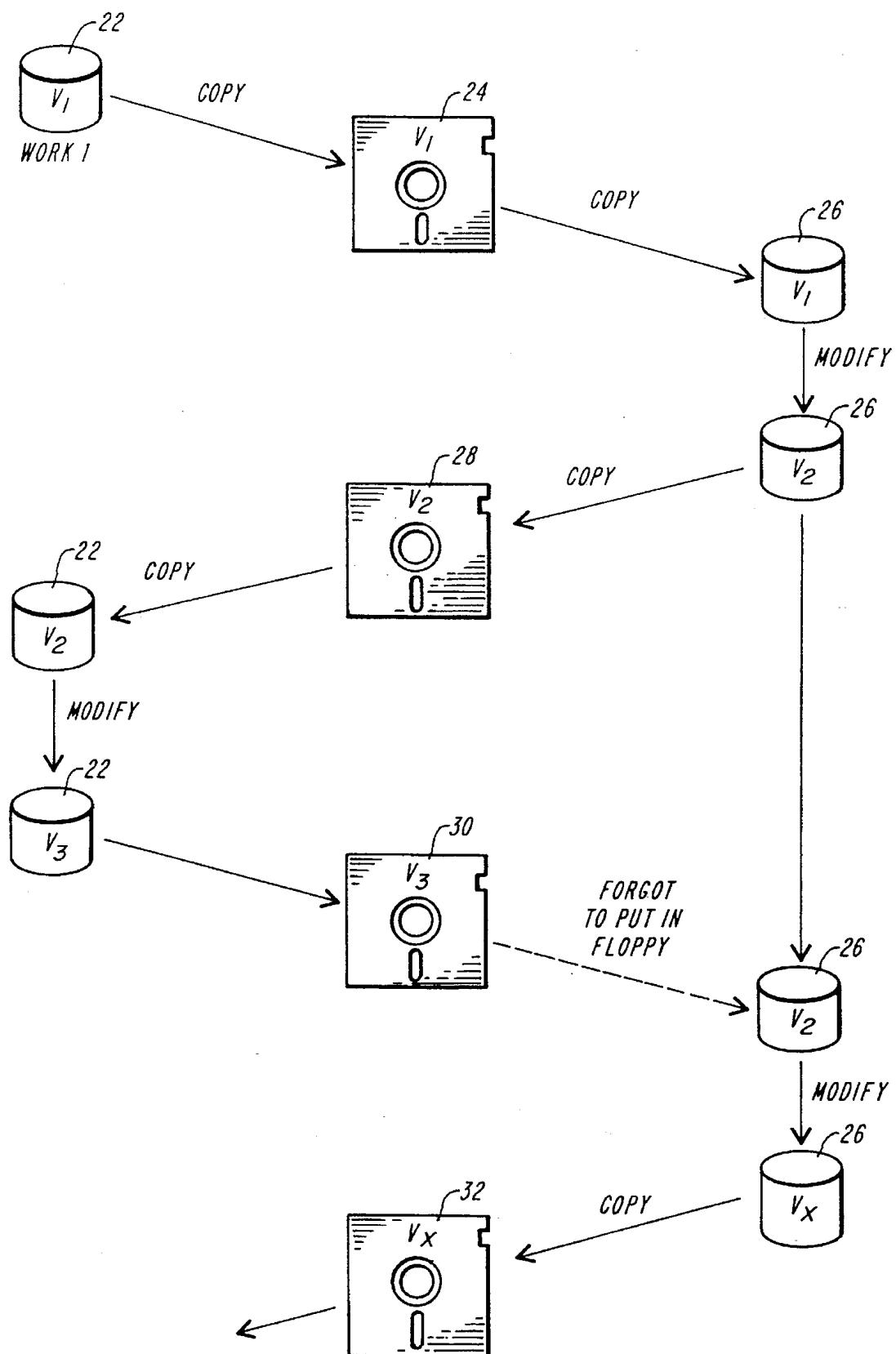


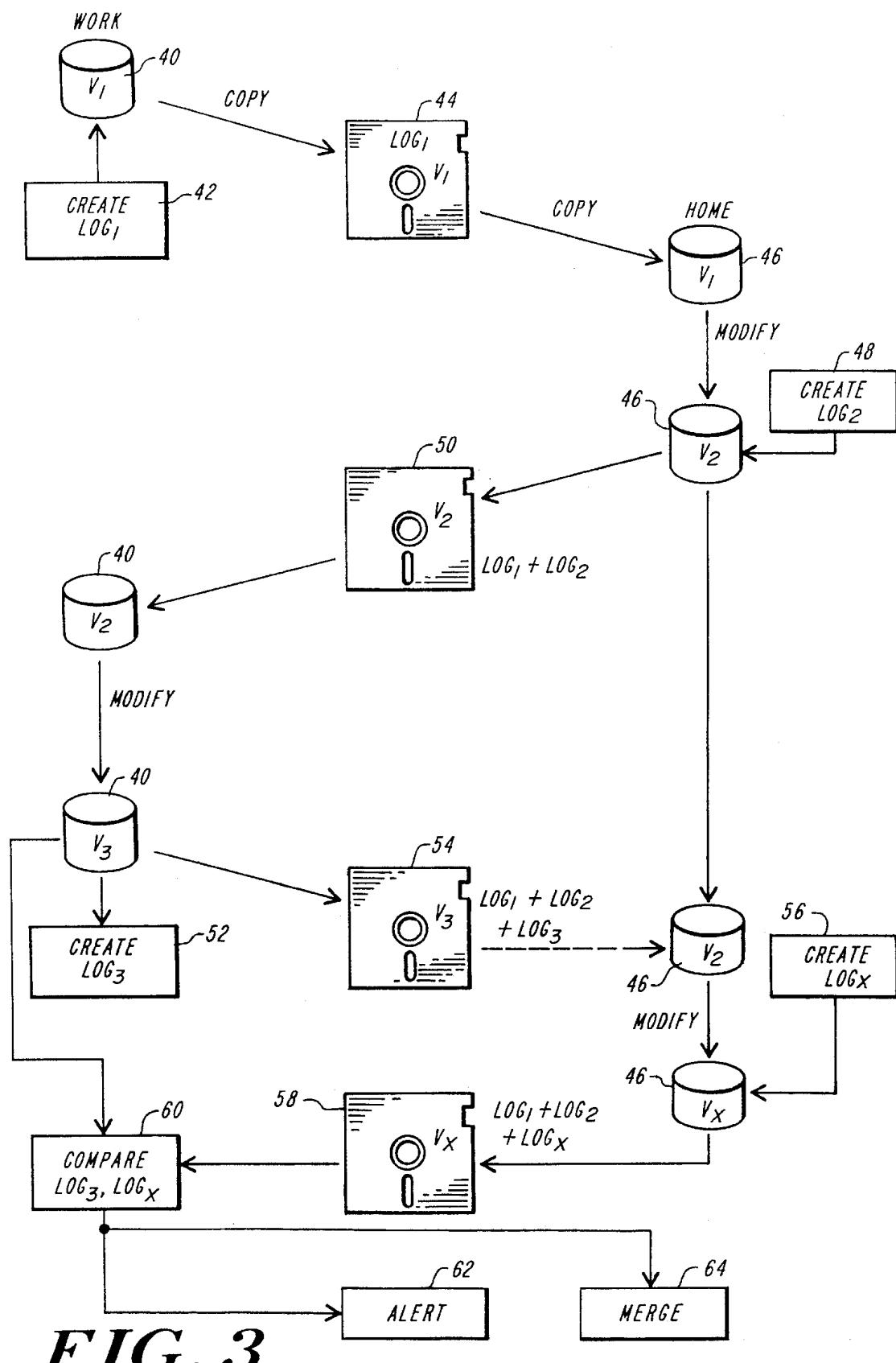
FIG. 2

U.S. Patent

Feb. 4, 1997

Sheet 3 of 5

5,600,834

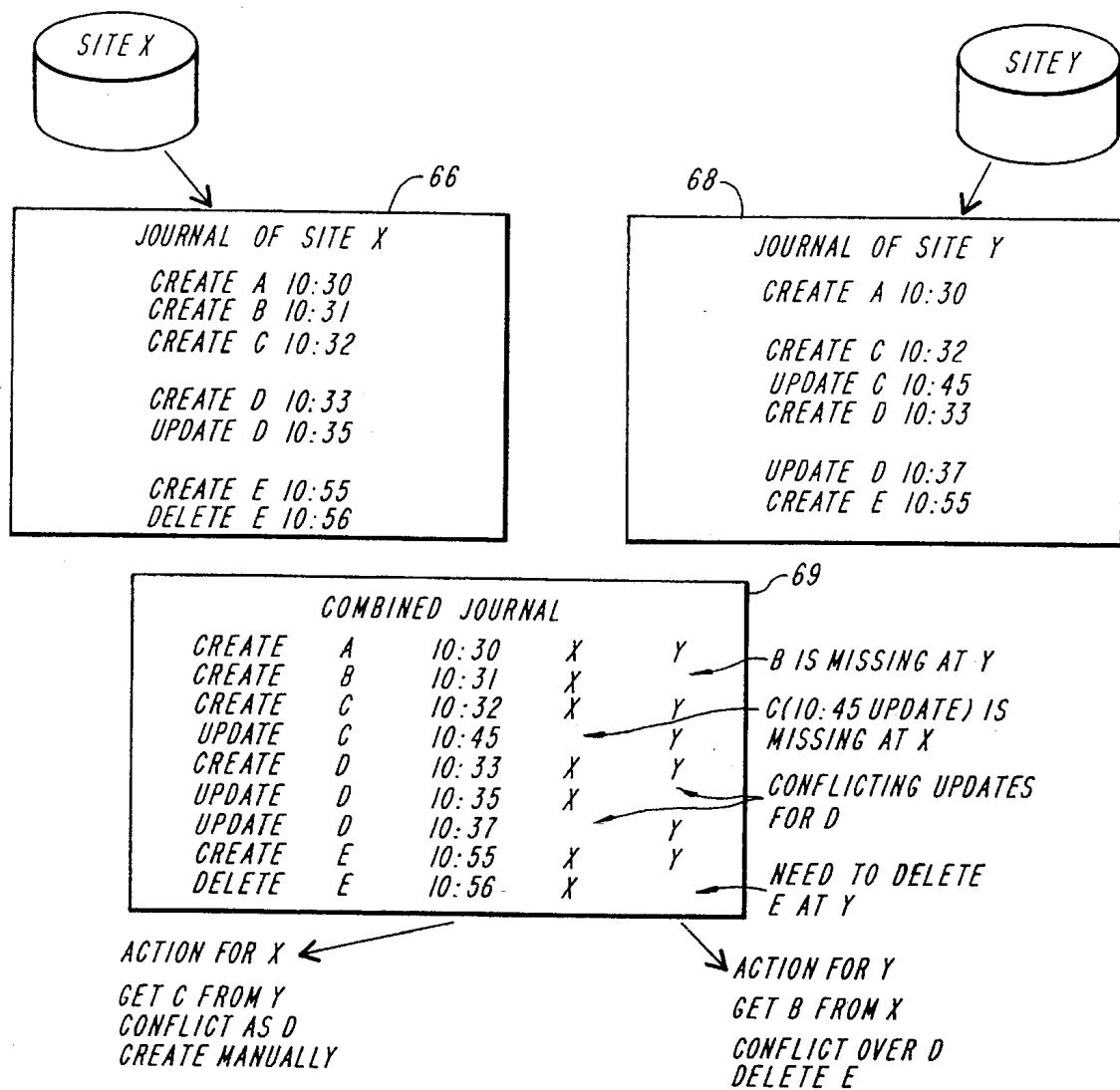
**FIG. 3**

U.S. Patent

Feb. 4, 1997

Sheet 4 of 5

5,600,834

**FIG. 4**

U.S. Patent

Feb. 4, 1997

Sheet 5 of 5

5,600,834

	OPERATION	FILE	TIMESTOP	HOME	SITE	OFFICE
JOURNAL	CREATE	A	4/19/93 10:22	✓	✓	
	UPDATE	A	4/20/93 8:30	✓	✓	

UPDATE NEEDED : COPY FROM HOME TO WORK

CREATE	A	4/19/93 10:22	✓	✓
UPDATE	A	4/20/93 8:30	✓	→○

DELETE NEEDED : DELETE COPY AT WORK

CREATE	B	4/19/93 10:35	✓	✓
DELETE	B	4/20/93 8:15	○←	✓

CONFLICT DETECTED

CREATE	C	4/19/93 10:35	✓	✓
UPDATE	C	4/19/93 11:00	○	✓
UPDATE	C	4/20/93 8:30	✓	○

FIG. 5

**METHOD AND APPARATUS FOR
RECONCILING DIFFERENT VERSIONS OF
A FILE**

This is a continuation of application Ser. No. 08/061,674 filed on May 14, 1993, now abandoned. 5

FIELD OF INVENTION

This invention relates to distributed file systems and more particularly to a method and system for reconciling different versions of files, in which the files are stored in computers at two or more separate locations or sites. 10

BACKGROUND OF THE INVENTION 15

There is a problem, especially with the portability of computers and floppy disks that a given file, for instance, in a lap top may not reflect the same information or data as the same file at a desktop or fixed work station. 20

This is because work is frequently taken from location to location. As frequently happens, a file created at a fixed work station at the office may be modified at a remote location, such as one's home, by merely transporting a disk or diskette containing the file and modifying it at the remote location. 25 Multiple versions of the same file can also exist in distributed networks when files are modified or manipulated by multiple users.

Problems thus arise when the versions of the file at two sites, such as home and office, do not agree because they have not been identically updated. This can occur by accident when one forgets to transport a floppy disk from one location to the other; or when one forgets to load the disk altogether. 30

It is of course desirable to have some synchronization between versions of the same file when created or modified at two different sites. For instance, it is possible to have the same version of a file at two sites and only access one at a time. When, however, versions of a file are created at two sites, it is important to be able to update or reconcile the files at both sites so as to appropriately update both files, or only one file. 35

In the past, systems have compared the times that a file was updated at different sites, have automatically selected the most recent version, and have copied this version into the appropriate file at both sites. Such systems include the Novell, Netware, Sun Microsystems Network File System (NSF) and Andrew File Systems. All of these systems have problems with their automatic updating procedures. 40

It is also a feature of NFS, Andrew File System, and, Netware that they automatically alter files immediately after they are modified. This results in significant performance problems as new versions of files are transmitted. Moreover all updates are distributed throughout the network, exposing raw work product to all on the system. It can also be an embarrassment because of the automation process, where those connected to the distributed system immediately have knowledge of new unedited data and changes. 45

It will of course be appreciated that when there are multiple users or contributors to a single file, such as in writing software, or as in editing documents, it is very important to alert all users of the same file as to what others are doing so that at some point there is control in each of the users as to what updating or reconciling of multiple versions of the file will be permitted. It is particularly annoying for the writer of software to have someone else edit his software 60

without his knowledge. Likewise, it is equally unfortunate for the word processing public to have one user edit a work without giving adequate notice to the other user.

More specifically, an inadequate solution to the problem of multiple versions of the file at different locations exists in distributed file system technology as represented by the NFS, Andrew, Apple Share, Novell, and Research Systems software such as Coda and Ficus. All of these systems give the impression of being a single global file system. The advantages of having a single global file system are automatic updating, sharing, and familiar time sharing systems semantics. However, the problems with such systems are that they fail or degrade when disconnected, are unpredictable in performance, are unacceptable in that updates are at the system's convenience and not at the user's, and that they require a modified operating system, often requiring a single vendor.

Another inadequate solution to the problem of multiple revisions of a file is found in the explicit file transfer technology associated with diskette/tape, E-mail, Lap-Link and file transfer protocols. What these systems attempt to do is copy files and carry or mail them. While the advantages are complete user control, flexible transport, and conversion between different systems, the disadvantages include complicated and error-prone protocols, in which overwriting of useful data can occur accidentally and in which there are no "merges" of different versions. 20

In all these systems, the most recent version of the file in one computer is automatically copied to the other. Thus, current programs seek to establish which file is correct by date and time, a technique called "time stamping". However, these types of systems are far from failsafe. For instance, assuming one wishes to delete a file on a lap top, deleting the file at the lap top may not result in deleting the file at the fixed work station, but rather in restoration of the obsolete file found at the work station. Thus automatic reconciling systems are error-prone. 30

More generally, if some work is to be accomplished on a file in more than one place, then it is possible that neither supercedes the other. Time stamp based reconciliation thus will possibly result in over-writing relevant information. As a result, user's work embodied in the older version may be lost without any warning. It is also possible that this will only happen when one forgets to hook up the computers for the reconciliation between versions of the file. 40

What is important is to know when a file has been edited in two places, what has been done, whether or not to authorize a merge of the two versions, and on what basis. It is therefore important to devise a system by which a merge is done in a safe way. It is also important to provide a system in which conflicts are recognized, with the conflict not necessarily being resolved automatically, but rather at the option of an individual operator who has been alerted to the fact of a conflict. 50

Note that one prior art way of determining a conflict is the so-called "journaling" technique which is to keep a record of what has transpired at one central location. Using a single centralized computer, a forward log or journal type of reconciliation may be accomplished. 55

SUMMARY OF THE INVENTION

However, rather than keeping a centralized journal, it is a feature of the present invention that each computer or system keep its own journal. The journal, which is a history of file versions, indicates the file which is edited and its 65

date/time stamp. Optionally the journal may also keep a detail of the type of editing that was involved should a conflict be determined.

For reconciliation, if the files are the same and the journals agree, there is no conflict.

On the other hand, when one works on one computer but not the other, and the resulting files are subsequently to be merged together, the Subject System first compares the two journals to see if one has more journal entries in one than the other. Note the comparison may be facilitated by in a merge operation. Once having determined that there are differences in some of the journal entries, then the system automatically copies those files for which the journal indicates no conflict, and alerts the user so that actions can be taken to resolve any conflict found.

Different versions of the same file are thus reconciled by each computer maintaining its own journal and by the comparison of the two journals at times specified by the user, with the reconciling system automatically updating file revisions when appropriate, or providing the user with an indication that such automatic updating is inappropriate.

Specifically, the Subject System can be configured to either delete a file which has been determined to be the non-desired file, or to copy the most recent file, in a replacing operation, into the computer which does not have the most recent file. At some point the journals of both of the computers will be in synchronization. Thereafter if no journals change, there need be no indication made to the user that a conflict exists. If one of the journals changes at only one site, then it is possible to simply instruct the machine at the other site on command to do the same actions. However, if both journals are changed, it is very important to alert the user that a conflict cannot be resolved.

Note, in the Subject Invention, not only does reconciling include the concept of copying or deleting, one can increase the level of detail of the individual entries in the logs that are filed to alert the user that a simple merge/purge performed on a time stamp basis will not work. For example, if the user is warned, the user may run a program called DIFF which highlights the differences between the two files. At that point, the user may decide which of the two files he prefers or which changes should be made in what file.

Thus, in the Subject Invention, in a distributed file system, instead of giving the user the impression that there is only one set of files, the system provides user with the impression that there are different versions of a file which must be occasionally reconciled, although only at the convenience of the user. The Subject System solves the problem of multiple versions of the same file by reconciling on demand. Each computer has a local version of the same data, reconciled by comparing journals of local changes, with user intervention being called into play if conflicting changes are discovered.

Applications for the subject reconciling system include file cataloging and reconciliation, office applications and database management systems. Hardware can involve organizers, palm tops, pen based tablets and notebooks.

Further applications of the Subject System include merging records within files. Moreover, it is possible to batch update by exchanging journals.

It is an important feature of the subject invention that the reconciliation may be invoked under user or application control, either at the beginning or end of a working session or overnight, for instance.

In summary, in a distributed file environment, a system for safely updating a file without risk of losing work performed

at one site due to work performed on the file at another site includes maintaining a journal or log at each site which is updated after a file is modified. This journal is compared with the logs from other sites before a file is used at any one site, so that new versions can be propagated automatically and safely to out-of-date sites, with the user immediately alerted if conflicting versions of the file exist at different sites. Different versions of the same file are thus reconciled by each computer maintaining its own journal and by the comparison of the two journals at times specified by the user, with the reconciling system automatically updating file revisions when appropriate, or providing the user with an indication that such automatic updating is inappropriate. The reconciliation can be applied to collections of files, automatically updating only those files for which it is safe to and necessary do so. Since reconciliation occurs at times selected by the user, inconsistent or partially completed versions of files need not be propagated to other sites. Additionally, logs may be built incrementally by occasionally observing the state of the systems in terms of the files and their time stamps and creating additional log entries reflecting appearance, disappearance and changes of files. Furthermore, logs may be purged of obsolete entries by including additional log entries indicating the most recent time each site has participated in a reconciliation and deleting obsolete entries that all sites have seen.

BRIEF DESCRIPTION OF THE DRAWINGS

These and other features of the Subject Invention will be better understood taken in conjunction with the detailed description in conjunction with the drawings of which:

FIG. 1 is a diagrammatic representation of the transfer of versions of a file from an office computer to a remote location, for instance, in the home;

FIG. 2 is a diagrammatic representation of the problem of generating two different versions of the same file through the modification of the file at two different locations;

FIG. 3 is a diagrammatic representation of a method for reconciling versions of a file including the creation of logs at various sites and the comparison of the logs prior to permitting either automatic updating or manual updating when a comparison between the logs indicates a discrepancy;

FIG. 4 is a block diagram illustrating the generation of journals at two sites, forming a combined journal, detecting a conflict, and providing actions based on conflict resolution; and

FIG. 5 is a series of diagrams indicating journals and situations where no update is needed, an update is needed, a delete is needed, or a conflict is to be indicated.

DETAILED DESCRIPTION

Referring now to FIG. 1, in a typical operational setting, Version #1 of a file may be created at a fixed terminal 10 at an office 12 in which the version created at terminal 10 is stored in a storage device 14. The information in this file may be transferred as via a floppy diskette 16 to a computer 18 at a home site 19, with the file being modifiable so as to produce a Version #2 which is stored in a storage device 20.

Referring now to FIG. 2, it will be seen how it is possible to modify files at two different sites or locations such that work performed in the file may not necessarily be on the most recently updated file. As can be seen, Work 1 in storage device 22 contains the Version #1 which is copied onto a

5,600,834

5

diskette 24 that is then copied into storage device 26 at a remote site or location. The information in storage device 26 may be modified so as to produce a Version #2 at 26 which is then copied, for instance, onto a diskette 28 that is copied into storage device 22 at the first site as Version #2. This version may be further modified and placed in storage as Version #3. Thereafter this version may be copied onto diskette 30 which is intended to be downloaded to storage device 26. However in the process either the disk is lost or not downloaded, at which point the modification as illustrated at V_x is copied onto a diskette 32. It will be appreciated that there is now a problem in that Version #3 is different than Version V_x , which was created by modifying Versions V_2 as opposed to Version V_3 . This creates an error which is difficult to rectify and may be unnoticed.

The problem, of course, is that these are two versions of the same file. The first version will be Version Three and the second version will be Version X. Merely updating one of the computers with one version or the other will not solve the problem of reconciliation, because Version X does not have the updates of Version Three. Thus it is impossible to automatically update either of the versions at either of the different locations; and it is for this reason that time stamp based systems for reconciliation fail.

Thus, in terms of a typical scenario, considering the case of writing a book using personal computers at office and home, carrying files back and forth on a diskette, the normal procedure is to copy all the working files from the diskette to the computer about to be used, edit one or more chapters, and copy the edited files back to the diskette when done.

As a result one has three different copies of the files, one stored in the office computer, one at home, and one on the diskette. Even though there are really three copies, one thinks of them as being different versions of the same files.

If one forgets to copy the files edited at the office, one can then go home with an out-of-date diskette. The diskette carried home is then copied to the home computer and editing continues, not noticing that one is starting with stale information. The next day the updated files are copied back to the office computer, losing the previous work.

There are some things one can do to help protect against this common error. For example, some file copying programs have an option to check dates and refuse to replace a newer version of a file with an older one. This helps considerably, but is not perfect. It does not detect the error described above, for example, since the versions of the files edited at home in the evening do have a later date than the versions edited yesterday at work. It also fails to handle the case of deleting obsolete files.

Referring now to FIG. 3, the subject file reconciling system solves the above problems by embedding a program called RECONCILE in a system which detects conflicting updates, so one can use it to update files safely. The system will replace a file with a later version only if it is sure that the later version was derived from the one being replaced. If the file to be replaced is not an earlier version, the system will report an error so that one can resolve the conflict.

More specifically, assuming that a Version #1 of a file is stored at a storage device 40, the Subject System creates a log, Log_1 here illustrated at 42. When Version V_1 is copied onto a diskette 44, Log_1 also appears on the diskette. This diskette may then be loaded into a storage device 46 at a remote location, where the Version #1 storage may be modified to produce Version #2, which is again stored at storage device 46. Concomitant with the modification of Version #1 to Version #2, a further log is created, Log_2 as

6

illustrated at 48. When this file is to be transferred to the work location, it is downloaded to a diskette 50 which contains not only Version V_2 , it also contains Log_1+Log_2 , as illustrated. This is downloaded back to storage device 40 at the original work location which may be modified as illustrated by Version V_3 , again stored at storage device 40. Upon modification of the V_2 version, the system creates an additional log, Log_3 as illustrated at 52.

Again, when this version of the file is to be transferred to the remote location, it is downloaded to a diskette 54 which then contains not only Version V_3 , but also $Log_1+Log_2+Log_3$. This diskette, however, in the example given is not downloaded to storage device 46. Rather, as inadvertently sometimes happens, V_2 is modified to produce Version V_x . At the same time that V_x is formed, a log, Log_x , is created as illustrated at 56. Version V_x may ultimately be transferred to a diskette 58. This diskette will have Version V_x downloaded to it plus $Log_1+Log_2+Log_x$. If diskette 58 is then to be loaded back into the storage device 40, upon accessing of this file a unit 60 compares the logs previously generated at the work site, with the logs associated with diskette 58 which has been loaded at the worksite. The result of the comparison step is either to alert the operator at 62 to a difference in the logs for this file which will not permit automatic updating, or permit automatic updating as indicated by merge 64.

This being the case, a system is provided through the comparison of logs to either permit automatic updating or to alert the user that automatic updating is inappropriate.

In the scenario of FIG. 3, neither the office nor the evening versions of the files were derived from the other, so RECONCILE will prevent the system from overwriting them. Note the versions were both derived from the same earlier version, but not from each other.

The system knows when one version of a file was derived from another by keeping a history of past versions of files. If one history indicates that a file has gone through Versions #1, 2 and 3 while the other has only Version #1 and 2, it is safe to copy Version #3. But if one history shows Versions #1, 2 and 3 while the other shows Versions #1, 2 and 4, there is a conflict since neither Version #3 nor Version #4 was derived from the other.

More specifically, FIG. 4 shows how two journals are reconciled. Starting with the two separate journals, for Sites X and Y, here illustrated at 66 and 68 respectively, each journal or log contains entries describing the history of five files, named A, B, C, D and E. In addition to the file name, the journal entries indicate the action which was taken, either Create, Update or Delete, and the time which that action was taken, at that particular site. For example, the journal of Site X shows that file E was created at 10:55 and deleted at 10:56. Note, only times are shown for convenience, since the log typically indicates both time and date.

Note that the journals are ordered by file name: A, B, C, D and E; and by timestamp for the same file name. Journals are combined by merging them according to this rule: Identical entries (including the action taken) are combined into a single entry during the merge. The combined journal as illustrated at 70 also records which sites had each entry. This could be X, Y, or both X and Y. For example, the combined journal shows that E was created at 10:55 known to both X and Y, and deleted at 10:56, known only to Site X.

The goal of reconciliation is to bring the individual journals up to date by performing missing actions. Thus, in FIG. 4, the missing creation of file B at Site Y can be fixed by copying file B from Site X to Site Y. The missing update

of C at X can similarly be fixed by copying C from Site Y. In the case of file E, the missing action is a deletion, which can be corrected by deleting the copy of file E at Site Y. As these reconciliation actions are taken, the missing journal entries are filled in and the individual journals updated.

There is a conflict in the case of file D. Both sites agree that the file was originally created at 10:33, but they show independent updates occurring at different times, and neither site knows about the other site's update. The automatic reconciliation procedure reports this conflict rather than replacing either version, leaving it up to the user to perform whatever correction or merge of the individual files is necessary.

By way of further description, the following definitions are useful in understanding the Subject Invention.

For purposes of this invention, a file is a body of closely related information stored in a computer. Typical examples of files would be documents edited with a word processor, or spreadsheets, or messages. Each individual memorandum, letter, or book chapter is kept in its own file. In addition to its contents, a file has a name and a timestamp. The name identifies the file in general and the timestamp indicates when the file was created or changed. As time passes, a file with the same name will have different versions, which can be distinguished by their different timestamps.

A directory is defined as a collection of files. Usually the files in a directory have some loose relationship, for example that they are all part of some larger body of information like the chapters in a book; or that they were created by the same person, relate to the same topic, or are owned by the same organization. Directories also have names, and may also have timestamps, although directory timestamps are not very useful.

Most computer systems allow files and directories to be arranged in a hierarchy or tree, which means that directories can contain subdirectories. An advantage of subdirectories is that more closely related files can be grouped together. To find a file one works one's way into the successive subdirectories until the desired file is reached.

A working session is defined as a period of work on a single computer. During the course of a working session, files may be in an incomplete or inconsistent state. One ordinarily doesn't want to make a permanent record of these files or to send copies elsewhere. Usually one tries to finish a day's work by cleaning up the inconsistencies before ending the working session, although occasionally a session may last several days. Note, a session can be anything one chooses. It is, however, important to note that one doesn't use RECONCILE to copy files during a working session, but only at the beginning and/or end.

A site is a specific storage location for a directory hierarchy. As understood herein, several sites are considered as all containing versions of the same directory hierarchy. These versions may be the same or different. The basic purpose of the Subject System is to combine hierarchies at different sites, making them all the same by safely updating versions of individual files.

One should not think of a site as being the total disk storage on any one computer. Usually a site would contain a number of unrelated hierarchies defined according the user's convenience. A personal computer, for example, might contain separate hierarchies for system software, installed applications, and one or more individuals' working files. While actual systems often glue these into a single super-hierarchy, it is easier to think of them as being separate.

A site may also be nothing more than a diskette. In fact, the way one copies files to and from the diskette in the above scenario is to reconcile the diskette version with the com-

puter at home or office that one is copying to or from. At the beginning of a working session, the Subject System will detect newer files on the diskette and copy them to the computer. At the end of the session, the Subject System will detect newer files on the computer and copy them to the diskette.

By way of definition, a journal is a history of file versions. To do its work, the Subject System creates a journal for each site, merges them to look for missing versions, and either updates by copying more recent non-conflicting versions, or else reports errors if there are conflicts.

As with database journals, the journals used by the Subject System contain not only names and timestamps but also actions. For this system these are very simple: either "update" or "delete", which can be inferred from the fact that a previously-present file has disappeared.

Including deletion operations in journals means that reconcile can safely propagate deletions to other sites, again checking for conflicts.

There are actually two kinds of journals: internal and external. An internal journal is stored as a special file within the directory it describes. In each hierarchy, each directory has its own internal journal. An external journal contains the same information, but has been extracted into a separate file, and stored somewhere else. Although this system can use both kinds, it can only update to or from internal journals. External journals may be used as sources of information about necessary updates, but the actual files and directories involved are not directly accessible.

One implementation of the Subject System is described in the version of RECONCILE attached hereto as Appendix A.

The simplest and standard way to use RECONCILE is to apply it to several directly accessible sites such as mounted disks or diskettes. For example the command

reconcile . a:\
35 would reconcile the current working directory (named ".") in most systems) with the diskette in drive A. The order of the two parameters doesn't matter. In this scenario, RECONCILE would be run when one begins using either the office or the home computer, and again at the end. So long as one never forgets to do this, all updating is automatic. One can even delete obsolete files without having them "come back" at the other computer.

Suppose one does happen to forget to reconcile at the beginning or end of a session, and one then updates some file. The next time one reconciles with the two conflicting versions of the file, one will obtain the error message:

reconcile: Conflicting Versions, ./oops and a:oops
At this point the two users should consult their memories of what the conflicting updates were, or use a tool such as diff to find and display the differences between the two versions. One now edits one or the other to merge changes, if necessary. Finally, the resulting good version of the file is copied to the other site, replacing the bad version there, usually a copy program which copies the timestamp as well as the contents. This will leave a record of the conflict in the journals, but since there is a more recent, non-conflicting version at both sites, RECONCILE will not indicate any conflict.

In one embodiment, the Subject System builds journals by comparing the actual directories with the previous versions of its own journals each time it is run. This means that it makes sense to run RECONCILE even for a single site:

reconcile .
This updates the internal journal of the current working directory. If one makes several successive versions of a file, RECONCILE will only see the last one since the last time it was run. This can actually be an advantage since the other

5,600,834

9

versions are of no particular significance as long as they are not transmitted to any other site.

One can choose how often one wants to run RECONCILE. Even if one forgets to run RECONCILE at the end of a working session, one will not lose anything permanently. The cost of forgetting a reconciliation will be an increased probability of conflicting updates, needing manual intervention at a later time.

Other applications for the Subject System are as follows:

Supposing the joint writing of a research paper with a colleague, one stores the various sections of the paper in a directory to which each has access. Ordinarily both users communicate directly to avoid conflicting updates, but sometimes one of the users forgets. This is handled with RECONCILE. Each user makes a private copy of the entire directory. Assuming the directories are named -tom/paper, -dick/paper, and -public/paper, and that Tom is the user in question, before beginning a working session, Tom performs the command

reconcile -tom/paper -common/paper

At this point there may be conflicts. If there are, Tom may need to give Dick a call to resolve them. Having done Tom is sure that his working version of the paper is in agreement with the shared version. During the course of the work various sections might be temporarily wrong, or inconsistent with each other, but since this is just a working copy and not the public version, Tom is not concerned. Eventually Tom will be happy with the final version having proofread it, and checks it back in with exactly the same command as above.

COMMAND SYNTAX

The syntax of the "RECONCILE" command is

reconcile [options] [[-mode] (directory file)] . . .

If no directories or files are provided, the current working directory (".") is used.

directory	names a directory containing an internal journal and files
file	names an external journal describing some remote site

("—" refers to an external journal on standard input or output) mode is one or more of the following letters:

r	read the journal but don't write it
w	write the journal but don't read it
o	do not update files, only the journal

Option parameters are:

-q	work quietly, suppressing messages about actions taken
-n	do not update any files (except journals) regardless of -u flags
-h	print a helpful description of the command syntax
-a sitename	Abandon named site. Use this to forget about a site which is no longer in use. This allows the program to discard obsolete journal entries needed only for reconciliations with old sites. Sites are automatically abandoned after two months, with warnings being printed after one month.

As to environment, in addition to command line parameters, RECONCILE gets a name for the computer system being used from the environment variable \$HOST, using "UNKNOWN" if it is undefined.

10

As to the overall sequence of events, RECONCILE performs its processing in the following general steps:

1. Parse parameters, building a list of sites to be reconciled. If no sites are given, use "." (the current working directory) as the only site.

2. Read the old journal file for each site.

3. For each internal site, update the journal by examining the files currently present at that site.

4. Update the list of known sites and their most recent reconciliation times. Events known from these times to be obsolete at all known sites will be discarded.

5. Perform the actual reconciliation, detecting conflicts and replicating files when there is no conflict. Update the internal journals accordingly.

15 6. For each site, write out an updated journal file.

These steps are described in more detail below.

As to the parameter parsing step, Step 1, this is a straightforward process of examining the parameters sequentially. It is performed by procedure "main" in module "reconcile."

20 For reading and writing journals as in Steps 2 and 6, journals read by procedures "readjournal" and "readentries" in module "journal", and write by procedures "writejournal" and "writeentries". The journal file format is editable text, described in the Journal File Formal section below.

25 As far as updates from the actual directory as in Step 3, journals are brought up to date with reality by reading the actual directory and inserting journal entries accordingly. This work is done by procedure "readdir" in module "journal." Current directory entries are considered one at a time. For each one found, a new journal entry for the site, time, and filename is created if none exists, and the new or existing entry is marked as having been confirmed.

30 After the entire directory has been read, a pass is made through the journal looking for unconfirmed entries. An unconfirmed entry indicates that a file once existed but no longer does; that is, that it has been deleted. For each such unconfirmed entry, a new journal entry is created with a deletion action and the current time as its timestamp.

35 For reconciliation as in Step 5, the actual reconciliation is performed on the internal representation of the journals by procedure "reconcile" in module "journal."

40 Reconciliation is performed only for the current entry; that is, the most recent entry for the file. Previous entries refer to out-of-date versions of the file. The goal is to make the entry and file present at every site. For each site at which the entry is not present, the program first checks to see if there is a conflict, see below. If there is none, the file is then copied from any site at which the current entry exists. Copying the file may actually be deleting it if the current event is a deletion event, or may involve creating or deleting a directory or symbolic link. Copying and deleting is done by procedures in the site module, since the action to be performed may depend on the type of site.

45 Detecting inconsistencies is accomplished as follows. A conflict exists for a site if the site does not have the current version of a file, but does have some previous version, and the current version is not derived from the previous one, as defined below. Existence of derivation shows that the current version came from the previous one by a connected series of user actions. This implies that it is safe to replace the prior version with the current one, since doing so is equivalent to replaying the sequence of actions in the derivation. Lack of a derivation implies that the replacement may be unsafe and therefore should not be done automatically.

50 60 A derivation is a sequence of steps which convert the older version to the newer one. Each step is directly reflected in the journal of some site as a successive pair of entries, one

5,600,834

11

following the other, with no other entries for the file in between at that site. There might be intervening entries at other sites, which would mean conflicts elsewhere but not for the derivation being considered. This sequentiality at a particular site implies that the later of the two successive versions was created directly by editing or modifying the earlier of the two. A derivation is like an audit trail. As it records all the steps taken to convert the older version to the newer one.

Existence of a derivation is determined by procedure "connected" of module "journal", which follows the direct steps backward from the current version, skipping unrelated events, until it reaches the prior version.

With respect to discarding obsolete events (step 4), without some way of discarding obsolete events, the journal files would grow indefinitely. An event is obsolete at a given site if there is a more recent event for the same file at the same site. If an event is obsolete at all sites, it can be discarded because it will never cause an inconsistency.

In order to track this, journals contains a list of "known sites", each marked with time of the most recent reconciliation involving the known site. This list is propagated and updated as reconciliations occur. The known site times indicate when information came from the known sites. In addition, each known site reflects these time stamps back by generating a list of acknowledgements. An acknowledgement gives the name of the known site, the name of a site known to it, and the timestamp of that source site. Again, these acknowledgements are propagated and updated appropriately. Originating sites may use the timestamps of the acknowledgements to determine when events are obsolete at other sites. Any event which happened before the oldest acknowledgement time must have been propagated to all known sites, so its predecessors can be discarded without causing conflicts.

Finally, there is a potential problem about lost sites. If a site fails to produce acknowledgements, it will cause events to accumulate indefinitely. This could easily happen if the site is no longer used. To avoid this, the program issues warnings about any sites last heard from more than a month ago, and removes them from the known site list after two months. There is also a way to discard a known site immediately.

Note that as to Journal File Format, journal files are standard text files which can be observed and even changed with any text editor. The files are formatted with one line per entry, plus a header line.

The header line has the format:

Journal of <sitename> <date> <time>—
<programname>where <sitename> is the fully qualified name of the file hierarchy being journaled, <date> and <time> are the time the journal was written, and

10

15

20

25

30

35

40

45

50

12

<programname> identifies the specific program which wrote the journal.

The entry lines each contain a fixed set of fields, separated by tabs, in the format:

5 <verb> <date> <time> <name><type>

where:

<verb> is one of a limited set of literals denoting possible actions:

+—create or update, making a new Version

—delete

*—some other site has a more recent Version, but it conflicts with the previous Version at this site.

<date> is the date the action occurred (yy/mm/dd format)

<time> is the time the action occurred (hh:mm:ss format)

<name> is the file's name, followed directly by the type (no intervening tab)

<type> is a single character;

(nothing) for an ordinary file

\ for a subdirectory

@ for a symbolic link

Supplementary lines give the date and time of the most recent reconciliation for each known site. Each line is of the form:

\$ <date> <time> <sitename>

Acknowledgements immediately follow the known site line for the acknowledging site, and are of the form

. <date> <time> <sitename>

where the acknowledging site (named by the immediately preceding known site line) is simply reflecting a reconciliation time back to its source.

Note that the RECONCILE program of Appendix A is built out of the following source modules, each of which is represented by a C++ source file (<name>.cpp) and a corresponding header file (<name>.h).

reconcil	Main program, parameter analysis, and user message generation.
site	Defines sites.
journal	Defines journals, performs most of the actual work.
timestamp	Defines internal and external formats for date and time information.
myalloc	Performs storage allocation and checks for memory leaks.
entry	Defines individual events.
filesystem	Performs local file system input/output operations.
knownsite	Defines known sites, manages times for deleting obsolete events.
parse	Supports text parsing operations in parameter and journal file processing.

The following is a program listing for the RECONCILE program, referred to herein as APPENDIX A.

5,600,834

13

14

APPENDIX A

5,600,834

15

16

5,600,834

17

18

3

```

switch (site.b) {
    // site.b = description of a site being reconciled with others
    // Define class Site
    // -----
    // ----- Site class iteration -----
    class Site {
        public:
            Site(BOOL mode, BOOL bmode); // constructor
            ~Site(); // destructor
            static void freeSite(void) // delete all sites
                {delete firstsite; }

            virtual BOOL isLocal(void) = 0; // is a local site
            BOOL isLocal(void); // provides reconciliation info
            (return (isLocal() || !writtenonly));
            BOOL isRemote(void); // receives reconciliation info
            (return (isLocal() && !readonly));
            BOOL isWritable(void); // OK to read journal
            (return !writtenonly);
            BOOL isWritable(void); // OK to write journal
            (return !writtenonly);
            virtual BOOL isUpdatable(void); // OK to update
            (return FALSE);

            void donotwrite(void); // prevent writing journal
            (readonly = TRUE);

            static Site* getFirstSite(void); // start iteration over sites
            (return firstsite);
            Site* getNextSite(void); // continue iteration over sites
            (return nextsite);

            void setTimestamp(const char* sitename); // set actual site name
            const char* getTimestamp(void); // get site name
            virtual const char* getRootName(void); // get local directory root
            (return (sitename != NULL) ? sitename : getRootName());
            incMask(void); // get site mask bit
            TimeStamp getLastAck(void); // get time of last ack
            (return lastAck);
            void setAckTime(TimeStamp t); // set time of last ack
            (lastAck = t);

            // JOURNAL FILE I/O
            virtual FILE* openJournal(const char* subdir, const char* mode) = 0;
            virtual FILE* complainaboutJournalFile(const char* subdir, const char* message, const char* buf);
            (buf[0] = 0);

        private:
            static Site* firstsite; // head of site list
            Site* nextsite; // part in site list
            const char* sitename; // site name - as consistent as possible
    };
}

```

```

switch (site.b) {
    // -----
    // timestamp.h - time stamping and comparison
    // -----
    // Declares class Timestamp.
    // Declares global variable 'now', time program is being run.
    // -----
    class Timestamp {
        public:
            // constructor
            Timestamp(void);

            // reset to very early time (can distinguish with neverset() below)
            void reset(void);

            // sec from a system time
            void stampTime(time_t systime);

            // extract a system time
            time_t gettime(void);

            // fill in date from text in buffer
            BOOL parsetime(const char* buff);

            // fill in time from text in buffer
            BOOL parsetime(const char* buff);

            // format date to returned string
            char* anameat(void);

            // format time to returned string
            char* atotime(void);

            // compare two timestamps (return -1, 0, +1, as if ts1>ts2)
            friend int compareTime(const Timestamp* ts1, const Timestamp* ts2);

            // set to now - months
            void getpast(const int months);

            // NOTE: if timestamp was reset, never set to an actual value
            BOOL neverset(void) { return (no == 0); }

        private:
            unsigned char yr;
            unsigned char mo;
            unsigned char dy;
            unsigned char hr;
            unsigned char mn;
            unsigned char sc;
    };
}
// ----- global -----
extern Timestamp now;

```

5,600,834

19

20

5,600,834

21

22

5

Filesystem 14.09.2014 filesys.cpp

filesys.cpp

```

if (infid < 0)
    return FALSE;
outfd = creat(targetname, ab.st_mode);
if (fdopen(outfd, "w") == NULL)
    return FALSE;
if (fseek(outfd, 0, SEEK_SET) != 0)
    FileIoWriteMessage("write", targetname);
close(infd);
close(outfd);
rmfile(targetname);
return TRUE;
}

getbuffer(buffer, buffsize);
for (i++) // copy loop
if ((n = readinfd(buffer, buffsize)) < 0)
    FileIoWriteMessage("read", sourcecname);
    close(infd);
    close(outfd);
    rmfile(targetname);
    return FALSE;
}

if (n < 0) // break;
while (n > 0) {
    rc = writeoutfd(buffer, n);
    if (rc < 0)
        break;
    n -= rc;
}
if (rc < 0) {
    FileIoWriteMessage("write", sourcecname);
    close(infd);
    close(outfd);
    rmfile(targetname);
    return FALSE;
}

close(infd);
close(outfd);
stat(targetname, &ab);
rewbuffer(buffer);
return TRUE;

// create new subdirectory
BOOL createsubdir(const char* targetname, struct stat& ab);
    BOOL saveproblem;
    if (!mksubdir(targetname))
        saveproblem = TRUE;
}

```

```

    // _MDO5_
endif // _MDO5_
else
    FilesystemMessage("create subdirectory", targetName);
    return FALSE;
}

SaveProblems = problems;
SetAttribute(problems, obj);
problems = SaveProblems;
return TRUE;
}

// read symbolic link

ifdef __MDO5__
#pragma argsused
endiff // __MDO5__
BOOL ReadLinkIn(char* sourceName, char* buff, int size, struct stat* sb)
{
    ifdef __MDO5__
        return FALSE;
    else // __MDO5__
        if (!GetAttributess(sourceName, sb))
            return FALSE;
        if (ReadLinkIn(sourceName, buff, size) < 0)
            FilesystemMessage("read link", sourceName);
        return FALSE;
    }
    return TRUE;
endiff // __MDO5__
}

// create symbolic link

ifdef __MDO5__
#pragma argsused
endiff // __MDO5__
BOOL CreateLinkIn(char* targetName, char* buff, struct stat* sb)
{
    ifdef __MDO5__
        return FALSE;
    else // __MDO5__
        if (!FormatIn(buff, targetName, 0))
            FilesystemMessage("create link", targetName);
        return FALSE;
    }
    SetAttributess(targetName, sb);
    return TRUE;
endiff // __MDO5__
}

// get attributes

BOOL GetAttributess(char* sourceName, struct stat* sb)
{
    if (stat(sourceName, sb) < 0) {

```

filesys.cpp

```

    FilesystemMessage("get attributes of", sourcename);
    return FALSE;
}
#endif TRUE

// set attributes
BOOL setattributes(const char* targetname, struct stat* ab)
{
    struct utimbuf ut;
    ut.actime = ab->st_atime;
    ut.modtime = ab->st_mtime;
    if (utime(targetname, &ut) < 0)
        FilesystemMessage("set date for", targetname);
    if (chmod(targetname, ab->st_mode) < 0)
        FilesystemMessage("set access control for", targetname);
    return TRUE;
}

// remove file
BOOL rmfile(const char* targetname)
{
    if (!unlink(targetname) < 0)
        FilesystemMessage("remove", targetname);
}
return TRUE;
}

// remove subdirectory
BOOL rmdir(const char* targetname)
{
    if (rmdir(targetname) < 0) // probably nonempty xxx
        FilesystemMessage("remove subdirectory", targetname);
    return TRUE;
}

// remove link
#ifndef _M600_
#define _M600_
    if (link(targetname, targetname) < 0)
        FilesystemMessage("remove link", targetname);
#endif // _M600_
    return FALSE;
#else // !_M600_
    if (!unlink(targetname) < 0)
        FilesystemMessage("remove link", targetname);
    return TRUE;
#endif // !_M600_
}

// ----- Pathname manipulation (defined in DOS, but not in Unix) -----
#ifndef _M600_

```

```

// convert a path name from relative to absolute
char *_fullpath(char* buffer, const char* path, int buffersize)
{
    int n;
    if (*path == PATHSEPARATOR) {
        if (buffersize >= 1 + buffersize)
            return NULL;
        else
            strcpy(buffer, path);
    }
    else {
        if (*path == '.' && *(path + 1) == PATHSEPARATOR)
            path++;
        if (*path == '/' && buffersize >= 1)
            return NULL;
        n = strlen(path);
        if (n + 1 + strlen(path) >= buffersize)
            return NULL;
        buffer[n] = PATHSEPARATOR;
        strcpy(buffer + n, path);
    }
    return buffer;
}

// split a pathname into its various components
void _splitpath(const char* path, char* drive, char* dir, char* name, char* ext)
{
    int c, sep, dot;
    if (path == NULL) path = "";
    for (c = 0; c < strlen(path); path[n] = '\0', n++) {
        switch (path[n]) {
        case PATHSEPARATOR:
            sep = n + 1;
            dot = 0;
            break;
        case '.':
            dot = n + 1;
            break;
        }
        if (dot == 0)
            dot = n + 1;
        dot += 1;
        if (drive == NULL)
            *drive = '\0';
        if (dot > 0 && path[n - dot] == '.')
            drive[n] = '\0';
        if (name == NULL)
            strcpy(name, path + sep, dot - sep);
        else
            name[dot - sep] = '\0';
    }
    if (ext == NULL)
        strcpy(ext, path + dot, n - dot);
}

// get one element from path name
BOOL getlpathelement(const char* scan, const char* element)
{
    char buff[256];
    char* stuff;
    if (scan == NULL)
        return FALSE;
    while (*scan != PATHSEPARATOR)
        scan++;
    if (*scan == '\0')
        *scan = '.';
    if (buff == NULL)
        buff = scan;
    else
        element = NULL;
    GETCOPY(element, buff);
    return TRUE;
}

// get disk label
void getlabel(const char* drive, char* label, int labelsize)
{
    static char* hostname = NULL;
    if (_getdrive(drive, "\\", "", ""))
        if (findirpath(drive, _ffblk, FA_LABEL) == 0 && (_ffblk.ff_attr & FA_LABEL) != 0)
            if (_ffblk.ff_label[0] != '\0')
                strcpy(label, _ffblk.ff_label);
}

// reconstruct file name from component parts
void _makepath(char* path, const char* drive, const char* dir, const char* name, const char* ext)
{
    int n;
    if (drive == NULL && Drive != '\0')
        FAILURE;
    if (dir == NULL && Dir == '\0')
        n = strlen(drive);
    strcpy(path, drive);
    path[n] = '\0';
    if (dir[n] != PATHSEPARATOR)
        *path++ = PATHSEPARATOR;
    if (name[n] == '\0' && name[0] != '\0') {
        if (n < strlen(name))
            strcpy(path, name);
        path[n] = '\0';
    }
    if (ext[n] == '\0' && ext[0] != '\0') {
        if (path[n] == '\0')
            strcpy(path, ext);
        else
            strcpy(path + n, ext);
    }
}

#endif // __MSDOS__


// ----- file name manipulation -----
// get fully qualified file name from directory and entry name
const char* getpathname(const char* dirname, const char* dirpath, const char* entryname,
    const char* sourcefile, const int sourcefilelen)
{
    const char* scan;
    const char* namestart;
    const char* pathname;
    if (dirname == NULL || dirname[0] == '.' && dirname[1] == '\0')
        pathname = copystring("", sourcefile, sourcefilelen);
    else
        pathname = copystring(dirname, sourcefile, sourcefilelen);
    if (dirpath == NULL) dirpath = "";
    if (entryname == NULL) entryname = "";
    if (*dirpath == '\0' && *entryname == '\0') {
        if (pathname[0] == '\0')
            _makepath(pathname, dirpath, entryname);
        else
            _getcopy(pathname, dirpath, entryname);
    }
    return pathname;
}

// ----- file name manipulation -----
// get file name from file block
for (i = 1; i < 0x100; i += sizeof(_ffblk)) if (_ffblk.ff_name[i] == '\0') i++ {
    if ((ffblk.ff_name[i] == '.' && i < labelsize - 1))
        label[i] = ffblk.ff_name[i];
    label[i] = '\0';
    return;
}
#endif // __MSDOS__
drive = drive; // prevent warning about drive not used
#endif // __MSDOS__


// disk label didn't work, use host name
if (hostname == NULL)
    hostname = getenv("HOST");
if (hostname == NULL)
    hostname = "UNKNOWN";
strcpy(label, hostname, labelsize);
label[labelsize - 1] = '\0';

}

```

5,600,834

25

26

丁

```

// journal.h
// journal.h - journal manipulation routines
// Defines class Journal, performs most major manipulations.
// -----
// ----- includes -----
#include "stdio.h"
#include "sys/types.h"
#include "sys/stat.h"
#include "ctype.h"
#include "errno.h"
#include "string.h"

#include "direct.h"
#ifndef _M6802
#define stat stat
#endif // _M6802

#include "sysalloc.h"

#include "reconstr.h"
#include "timeStamp.h"
#include "file.h"
#include "list.h"
#include "entry.h"
#include "journal.h"
#include "parse.h"
#include "filesys.h"

// ----- internal statics & declarations -----
static char *jheader = "Journal of:";

// ----- Journal code -----
// Journal constructor
Journal::Journal(const char* subdirectory)
{
    nextJournal = NULL;
    firstentry = NULL;
    subdir = subdirectory;
    writemask = 0;
}

// Journal destructor
Journal::~Journal(void)
{
    RELEASH(subdir);
    while (firstentry != NULL)
        deleteEntry(firstentry);
    while (nextJournal != NULL)
    {
        Journal *j = nextJournal;
        nextJournal = j->nextJournal;
        delete j;
    }
}

// ----- Journal methods -----
// add entry to journal
void Journal::addEntry(const Action action, const char* entryname, constTimeStamp ts,
                      EntryType type, int mask)
{
    Entry *e;
    Entry *p;
    int compare;

    compare = 0;
    for (p = firstentry; p != NULL; e = p, p = p->nextentry)
    {
        if ((compare = namecmp(e->entryname, entryname)) == 0)
        {
            if (e->type == type)
            {
                if (e->timestamp == timestamp)
                {
                    if (e->action == action)
                    {
                        if (e->mask == mask)
                            if (compare > 0)
                                break;
                    }
                }
            }
        }
    }
    // protect against inability to add directory timestamp
    // use timestamp creation time at this point if it is there
    if (mask & 0x40000000 && action == ADD)
    {
        if (p->mask & 0x40000000)
            p = NULL; // samecompentryname, p->entryname == 0)
        for (e = p; e != NULL && (e->mask & mask) == 0; e = e->nextentry)
            if (e != NULL)
                compare = 0;
    }
    if (compare == 0)
    {
        e = newEntry(action, entryname, ts, type);
        if (p == NULL)
        {
            e->nextentry = firstentry;
            firstentry = e;
        }
        else
        {
            e->nextentry = p->nextentry;
            p->nextentry = e;
        }
        e->preventry = p;
        if (e->nextentry != NULL)
            e->nextentry->preventry = e;
    }
    if ((e->mask & mask) == 0)
        e->mask |= mask;
    return e;
}

// delete entry from journal
void Journal::deleteEntry(Entry *e)
{
    Entry *p = e->preventry;
    Entry *n = e->nextentry;
    if (p == NULL)
        firstentry = n;
    else
        p->nextentry = n;
}

```

5,600,834

27

28

8

5,600,834

29

30

5,600,834

31

32

10

```

        knowni.cpp

        {
        ermsg = "wrong time format";
        return FALSE;
    }

    scan = getword1(scan, word, siestoword);
    k = makesondate(word);

    if (comparisonequals(k, known + k), tsi < 0)
        action = known + k, tsj;
    return TRUE;
}

else if (fscanf(scan + skipprefix("Ack", buff)) == NULL)
{
    scan = getword1(scan, word, siestoword);
    j = makesondate(word);
    scan = getword1(scan, word, siestoword);
    if (fscanf(scan + skipprefix("Ack", buff)) == NULL)
        ermsg = "wrong date format";
    return FALSE;
}

scan = getword1(scan, word, siestoword);
if (fscanf(scan + skipprefix("Ack", buff)) == NULL)
    ermsg = "wrong time format";
return FALSE;
}

ASSERTIVE = 0 <= k <= known);
internalack(1, 3, csj);

return TRUE;
}

else if (fscanf(scan + skipprefix("Ack", buff)) == NULL)
{
    scan = getword1(scan, word, siestoword);
    if (fscanf(scan + skipprefix("Ack", buff)) == NULL)
        ermsg = "wrong date format";
    return FALSE;
}

scan = getword1(scan, word, siestoword);
if (fscanf(scan + skipprefix("Ack", buff)) == NULL)
    ermsg = "wrong time format";
return FALSE;
}

scan = getword1(scan, word, siestoword);
j = makesondate(word);

ASSERTIVE = 0 <= k <= known);
internalack(1, 3, csj);

return TRUE;
}
else
return FALSE;
}

// acknowledge (internal version)
// indicate that site k knows about j up to time tk)

```

```

void KnownSite::internalAck(int k, int j, Timestamp tk)
{
    if ((comprelimes(ack[k] + known[j] + j), tk) < 0)
        ack[k] = known[j] + j + tk;
}

// acknowledge (name has received information from rname)
void KnownSite::acknowledge(const char* name, const char* rname)
{
    int k = makeknownsite(rname);
    int j = makeknownsite(name);
    internalAck(k, j, ack[k] + known[j] + j);
}

// this site is target of a reconciliation
void KnownSite::reconciliationTarget(const char* name)
{
    int k = makeknownsite(name);
    int j;

    ack[k] = known[k] + now;
    for (j = 0; j < known[k]; j++) {
        internalAck(k, j, ack[k] + known[j] + j);
    }
}

// kill site by name
void KnownSite::killSite(const char* name)
{
    int k = makeknownsite(name);
    known[k].killed = TRUE;
}

// kill old sites
void KnownSite::killOldsites(void)
{
    int k;
    Timestamp warndate;
    known[k].killdate = warndate;
    warndate.outcast();
    killdate.outcast();
    for (k = 0; k < known; k++) {
        if (comprelimes(known[k].killdate, ack[k] + known[k] + k) > 0) {
            known[k].status = "site is abandoned", known[k].valueone;
            known[k].killed = TRUE;
        }
        else if (comprelimes(warndate, ack[k] + known[k] + k) > 0) {
            WarningMessage("Site %s will be abandoned on %s", known[k].sitename, killdate);
            abandon[k] = TRUE;
        }
    }
}

// get time acknowledged by all known sites to given site
Timestamp KnownSite::getAckTime(const char* name)
{
    int k;

```

knownsit.cpp

5,600,834

33

34

5,600,834

35

36

5,600,834

37

38

12

```

reconcil.cpp //----- reconcil.cpp //----- reconcil.cpp

if (doReconcil)
{
    call();
    exit(1);
}

if (bdoosDirectory)
    makeDirectory(mode, umode, omode, "+");
}

// main - command line interface
void main(int argc, char* argv[])
{
    // setup
    problems = FALSE;
    nondate = FALSE;
    quiet = FALSE;
    paranoid = FALSE;
    parseArgs(argc, argv);
    if (problems)
        exit(0);

    // run the program
    if (quiet)
        fprintf(stderr, "Recon, version");
    doSites();
    if (problems && quiet)
        fprintf(stderr, "Successful reconciliation");

    // cleanup
    Site::exit();
    RemoteSite::exit();
    RemoteSite::reconcile();
    verifyEverything();
    exit(0);
}

//----- includes -----
#include "audio.h" // site.cpp - description of a site being reconciled with others
#include "adlib.h" // defines the Site class, subclasses LocalSite and RemoteSite.
#include "time.h" // Does all the actual I/O in doSiteLocSite.
#include "string.h" //----- Site code -----
Site* Site::firstsite = NULL;
int Site::nbsites = 0;

Site::Site(FILE* mode, BOOL umode)
{
    name = newname;
    newname-><< 1;
    if (name->val > 0)
        FatalMessage("Too many different sites");
    nextsite = firstsite;
    sitesite = NULL;
    writeonly = umode;
    readonly = mode;
    firstsite = this;
}

Site::Site(void)
{
    if (sitesite != NULL) RELEMEM(sitesite);
    if (nextsite != NULL)
        delete nextsite;
}

// define site name
// XXX needs to be fixed to make site names consistent
void Site::setSiteName(const char* sitenamearg)
{
    if (sitename == NULL)
        GCHCOPY(sitename, sitenamearg);
}

//----- LocalSite code -----

```

5,600,834

39

40

14

```

//----- RemoteSite code -----
RemoteSite::RemoteSite(const char* name, BOOL mode, BOOL weode, BOOL cmdode)
: SiteMode_(mode)
{
    char path[MAX_PATH];
    char drive[MAX_DRIVE];
    char dir[MAX_DIR];
    char file[MAX_FILE];
    char ext[MAX_EXT];
    char label[14];
    char result[MAX_PATH + 14];

    updatePath = cmdode;
    rootname = NULL;
    strcpy(result, name);
    fullpath(path, name, result);
    sprintf(path, "%s\\%s\\%s", drive, dir, file);
    sprintf(path, "%s\\%s\\%s.%s", path, ext);
    sprintf(result, "%s%s", label, path);
    setSiteName(result);
}

//----- LocalSite code -----
LocalSite::LocalSite(const char* name, BOOL mode, BOOL weode, BOOL cmdode)
: SiteMode_(mode)
{
    char path[MAX_PATH];
    char drive[MAX_DRIVE];
    char dir[MAX_DIR];
    char file[MAX_FILE];
    char ext[MAX_EXT];
    char label[14];
    char result[MAX_PATH + 14];

    updatePath = cmdode;
    rootname = NULL;
    strcpy(result, name);
    fullpath(path, name, result);
    sprintf(path, "%s\\%s\\%s", drive, dir, file);
    sprintf(path, "%s\\%s\\%s.%s", path, ext);
    sprintf(result, "%s%s", label, path);
    setSiteName(result);
}

//----- RemoteSite code -----
RemoteSite::RemoteSite(const char* name, BOOL mode, BOOL weode)
: SiteMode_(mode)
{
    jname = NULL;
    GENDRV(jname, name);
}

//----- LocalSite code -----
LocalSite::LocalSite(void)
{
    REMEM(jrootname);
}

// open journal file
FILE* LocalSite::openJournal(const char* subdir, const char* mode)
{
    FILE* journalfile;
    const char* pathname = GETPATHNAME(getrootname(), subdir, JOURNALFILE);
    journalfile = fopen(pathname, mode);
    REMADE(pathname);
    return journalfile;
}

// complain about contents of journal file
void LocalSite::complainaboutjournalfile(const char* subdir, const char* message, const char* buff)
{
    const char* pathname = GETPATHNAME(getrootname(), subdir, JOURNALFILE);
    WarningMessage("te %s: %s", pathname, message, buff);
    REMADE(pathname);
}

//----- RemoteSite code -----
RemoteSite::RemoteSite(const char* name, BOOL mode, BOOL weode)
: SiteMode_(mode)
{
    jname = NULL;
    GENDRV(jname, name);
}

//----- TimeStamp code -----
// timestamp.cpp - time stamping and comparison
// defines class TimeStamp
// -----
// ----- includes -----
#include <time.h>
#include <conio.h>
#include <timestamp.h>
#define YEARSTART 1900
#define DECADE 100

// ----- globals -----
TimeStamp now; // time of this program run

// ----- TimeStamp code -----
// create new timestamp
TimeStamp::TimeStamp(void)
{
    cswc();
}

// zero time stamp
void TimeStamp::reset(void)
{
    yr = 0;
    mo = 0; // out of range which is 1..12
    dy = 0; // out of range which is 1..31
    hr = 0;
    min = 0;
    sec = 0;
}

// convert system time to a timestamp
void TimeStamp::stampctime(time_t systemtime)
{
    struct tm *tm = localtime(&systemtime);
    yr = tm->tm_year - DECADE;
    mo = tm->tm_mon + 1;
    dy = tm->tm_mday;
    hr = tm->tm_hour;
    min = tm->tm_min;
    sec = tm->tm_sec;
}

// convert a timestamp to system time
time_t TimeStamp::gettime(void)
{
}

//----- RemoteSite code -----
RemoteSite::RemoteSite(const char* name, BOOL mode, BOOL weode, BOOL cmdode)
: SiteMode_(mode)
{
    if (jname == NULL) REMEM(jname);
}

// open journal file
FILE* RemoteSite::openJournal(const char* subdir, const char* mode)
{
    if (strcmp(jname, "") == 0) {
        autochar* jname;
        if ('T' == mode) return stdin;
        case 'W': return stdout;
        default: return NULL;
    }
    subdir = subdir; // prevent warning about subdir not used
    return fopen(jname, mode);
}

// complain about contents of journal file
void RemoteSite::complainaboutjournalfile(const char* subdir, const char* message, const char* buff)
{
    const char* pathname;
    if (strcmp(jname, "") == 0) {
        pathname = "(standard input)";
    } else {
        pathname = jname;
        WarningMessage("te %s: %s", pathname, subdir, message, buff);
    }
}

//----- LocalSite code -----
LocalSite::LocalSite(void)
{
    REMEM(jname);
    if (strcmp(jname, "") == 0) {
        pathname = "(standard input)";
    } else {
        pathname = jname;
        WarningMessage("te %s: %s", pathname, pathname, pathname, message, buff);
    }
}

//----- TimeStamp code -----
// timestamp.cpp - time stamping and comparison
// defines class TimeStamp
// -----
// ----- includes -----
#include <time.h>
#include <conio.h>
#include <timestamp.h>
#define YEARSTART 1900
#define DECADE 100

//----- TimeStamp code -----
// create new timestamp
TimeStamp::TimeStamp(void)
{
    unsigned char century, year, month, day;
    static const get3digit(unsigned char *value, const char *buff)
    {
        if (*buff < '0' || *buff > '9' || *(buff + 1) < '0' || *(buff + 1) > '9')
            return FALSE;
        *value = ((buff - '0') * 10 + (*(buff + 1) - '0')) * 10;
        return TRUE;
    }

    // get 3-digit number
    static const get3digit(unsigned char *value, const char *buff)
    {
        if (*buff < '0' || *buff > '9' || *(buff + 1) < '0' || *(buff + 1) > '9')
            return FALSE;
        *value = ((buff - '0') * 10 + (*(buff + 1) - '0')) * 10;
        return TRUE;
    }

    // get date from buffer
    bool timestamp_iparsedate(const char* buff)
    {
        unsigned char century, year, month, day;
        while (*buff == ' ' || buff == '\t')
            buff++;
        if (get3digit(&year, buff) && *(buff + 2) == '/')
            if (get3digit(&month, buff + 3) && *(buff + 5) == '/')
                if (get3digit(&day, buff + 6) && *(buff + 8) == '\0') {
                    century = 1900 - DECADE;
                    year = year + century;
                    month = month;
                    day = day;
                    record = TRUE;
                }
        if (get3digit(&year, buff) && *(buff + 2) == '/')
            if (get3digit(&month, buff + 3) && *(buff + 5) == '/')
                if (get3digit(&day, buff + 6) && *(buff + 8) == '\0') {
                    year = year + century + 100 - YEARSTART;
                    century = century;
                    day = day;
                    record = TRUE;
                }
        return FALSE;
    }

    // get time from buffer
    bool timestamp_iparsetime(const char* buff)
    {
        unsigned char hour, min, sec;
        if (get3digit(&hour, buff) && *(buff + 2) == ':')
            if (get3digit(&min, buff + 3) && *(buff + 5) == ':')
                if (get3digit(&sec, buff + 6) && *(buff + 8) == '\0') {
                    hr = hour;
                    min = min;
                }
    }
}

```

5,600,834

41

42

15

5,600,834

43

Having above indicated a preferred embodiment of the present invention, it will occur to those skilled in the art that modifications and alternatives can be practiced within the spirit of the invention. It is accordingly intended to define the scope of the invention only as indicated in the following claims. 5

What is claimed is:

1. In a computer system not utilizing a global database, but rather utilizing a collection of local databases, none of which serve as a central collection point, in which files are stored and modified at memory devices at multiple locations, apparatus for permitting the creation of new versions of a file at one location without knowledge of whether conflicting versions are created at another location, comprising:

a portable memory device transportable between locations on which a file version is stored; and,

means for safely permitting reconciling different versions of a file transported from location to location on said portable memory device, said reconciling means including

means at first and second locations for generating a log entry at its respective location, each of said log entries including a time stamp for the version of said file on said portable memory device at said location, said timestamp serving as a unique identifier regardless of the time indicated thereby, thus to create a history of version creation and modification for said file at said first and second locations;

44

means at said first location for entering said log entry into said portable memory device;

means at said second location for reading out the log entry on said portable memory device and for combining the log entry from said portable memory device and a log entry at said second location so as to compare both modified versions and respective timestamps to identify missing entries or confliction updates represented by different timestamps and thus the existence of different versions of said file;

means at said second location and upon said identification of different versions of said file for determining actions necessary to synchronize said different versions; and, means at said second location for determining if such synchronizing action would result in loss of information, whereby combination of all versions of said file may be safely accomplished at said second location.

2. The apparatus of claim 1, and further including means for performing the determined synchronizing actions by copying or deleting files.

3. The apparatus of claim 2, and further including means for notifying the user if said synchronizing actions would result in loss of information.

4. The apparatus of claim 1, and further including means for purging obsolete log entries by purging a first entry if there exists a subsequent entry for the same file and if that subsequent entry is available at all other locations.

* * * * *